Algoritmos

Búsqueda Lineal (Secuencial)

La **búsqueda lineal** o **búsqueda secuencial** es un algoritmo sencillo utilizado para encontrar un elemento en una lista o arreglo. Este algoritmo recorre secuencialmente cada uno de los elementos del conjunto de datos, comparando el valor buscado con cada elemento, hasta que se encuentra una coincidencia o se ha revisado todo el conjunto.

Este algoritmo es básico y fácil de implementar, pero no es eficiente para grandes volúmenes de datos, ya que su complejidad en tiempo es O(n), donde n es el número de elementos en la lista. Es particularmente útil cuando los datos no están ordenados o cuando la lista es pequeña.

Bibliografia recomendada:

- **Luis Joyanes Aguilar** en "Fundamentos de Programación" enfatiza su simplicidad y adecuación para listas pequeñas, presentando el algoritmo con un enfoque didáctico, utilizando pseudocódigo y ejemplos prácticos para la enseñanza.
- **Thomas H. Cormen et al.** en "Introduction to Algorithms" ofrece una explicación teórica detallada, abordando la complejidad del algoritmo y comparándolo con otros métodos de búsqueda, subrayando sus limitaciones en términos de eficiencia.
- **Unai López de la Fuente** en "Algoritmos y estructuras de datos en Python" presenta el algoritmo con un enfoque práctico en Python, proporcionando implementaciones y ejemplos para su uso en programas reales, además de discutir posibles optimizaciones.
- **Gustavo López et al.** en "Análisis y diseño de algoritmos" muestran implementaciones en C y Pascal, enfocándose en cómo el algoritmo puede ser codificado en estos lenguajes y su aplicación en situaciones prácticas, manteniendo un enfoque práctico y orientado a la implementación.

Ejemplo de Pseudocódigo de Búsqueda Lineal

```
Algoritmo BúsquedaLineal(lista, valorBuscado)
    Para i desde 0 hasta longitud(lista) - 1 hacer
        Si lista[i] = valorBuscado entonces
            Retornar i  // Se encontró el valor, retornar su    índice
        FinSi
    FinPara

    Retornar -1   // El valor no se encontró en la lista
FinAlgoritmo
```

Descripción del Pseudocódigo

1. **Entrada**: El algoritmo recibe una lista (arreglo) de n elementos y el valor que se desea buscar.
2. **Bucle Secuencial**: Se recorre la lista desde el primer hasta el último elemento.

3. **Comparación**: En cada iteración, se compara el valor del elemento actual de la lista con el valor buscado.
4. **Condición de Éxito**: Si el valor es encontrado, se retorna el índice donde se encontró.
5. **Condición de No Éxito**: Si el bucle termina sin encontrar el valor, se retorna -1, indicando que el valor no está en la lista.

Ordenamiento Burbuja (Bubble Sort)

El **algoritmo de ordenamiento Burbuja** (conocido en inglés como **Bubble Sort**) es uno de los métodos más simples para ordenar una lista de elementos. Este algoritmo trabaja comparando repetidamente elementos adyacentes y permutándolos si están en el orden incorrecto. Este proceso se repite hasta que la lista está completamente ordenada. Aunque es fácil de entender e implementar, Bubble Sort es ineficiente para listas grandes, con una complejidad temporal de $O(n^2)$, donde n es el número de elementos en la lista.

Bibliografia recomendada:

- **Luis Joyanes Aguilar** en "Fundamentos de Programación" describe el algoritmo Burbuja como un método intuitivo pero ineficiente para grandes volúmenes de datos. Presenta el algoritmo con un pseudocódigo claro y lo utiliza como una base para introducir conceptos de ordenamiento.
- **Thomas H. Cormen et al.** en "Introduction to Algorithms" explica el algoritmo desde una perspectiva teórica, abordando su complejidad y proporcionando análisis detallados sobre su funcionamiento. Comparan Bubble Sort con otros algoritmos de ordenamiento, destacando sus limitaciones y la existencia de métodos más eficientes.
- **Unai López de la Fuente** en "Algoritmos y estructuras de datos en Python" implementa el algoritmo en Python, mostrando cómo funciona en la práctica y ofreciendo ejemplos concretos de su uso. También se discuten posibles optimizaciones y su aplicación en situaciones específicas.
- **Gustavo López et al.** en "Análisis y diseño de algoritmos" presentan el algoritmo con implementaciones en C y Pascal, ofreciendo un enfoque práctico sobre cómo codificarlo en estos lenguajes, y discuten su relevancia histórica como un primer acercamiento al concepto de ordenamiento.

Ejemplo de Pseudocódigo del Algoritmo Burbuja

```
Algoritmo Burbuja(lista)
    Para i desde 0 hasta longitud(lista) - 2 hacer
        Para j desde 0 hasta longitud(lista) - 2 - i hacer// -i?
            Si lista[j] > lista[j + 1] entonces
                // Intercambiar los elementos adyacentes si están en el orden incorrecto
                temp ← lista[j]
                lista[j] = lista[j + 1]
                lista[j + 1] ← temp
            FinSi
        FinPara
    FinPara
FinAlgoritmo
```

Descripción del Pseudocódigo

1. **Entrada**: El algoritmo recibe una lista de n elementos que se desea ordenar.
2. **Bucle Externo**: Controla el número de pasadas necesarias, que es n-1, dado que en cada pasada el elemento más grande "burbujea" hasta su posición correcta.
3. **Bucle Interno**: En cada pasada, compara elementos adyacentes en la lista y los intercambia si están en el orden incorrecto.
4. **Intercambio**: Si el elemento actual es mayor que el siguiente, se intercambian sus posiciones para llevar el mayor hacia el final de la lista.
5. **Salida**: La lista queda ordenada en orden ascendente al final del proceso.

Ordenamiento Burbuja | Bubble Sort
Python - Nivel 31 - Reto 3 - Ordenamiento de burbuja
Python - Nivel 31 - Reto 4 - Burbuja mejorado

Ordenamiento por Inserción (Insertion Sort)

El **algoritmo de ordenamiento por inserción** (conocido en inglés como **Insertion Sort**) es un método de ordenamiento simple y eficiente para pequeñas cantidades de datos. Funciona de manera similar a cómo los humanos ordenan una mano de cartas: se toma un elemento y se coloca en la posición correcta dentro de los elementos ya ordenados. Aunque su complejidad temporal es $O(n^2)$ en el peor de los casos, es eficiente para listas que ya están casi ordenadas o que son pequeñas.

- **Luis Joyanes Aguilar** en "Fundamentos de Programación" presenta el algoritmo de inserción como un método adecuado para ordenar listas pequeñas o casi ordenadas. Explica el proceso paso a paso, usando pseudocódigo para ilustrar cómo se va construyendo la lista ordenada a partir de los elementos individuales, enfatizando su simplicidad y aplicabilidad en ciertos contextos.
- **Thomas H. Cormen et al.** en "Introduction to Algorithms" abordan el Insertion Sort desde un punto de vista teórico, explorando su eficiencia y comportamiento en diferentes escenarios. Discuten cómo el algoritmo funciona bien con datos casi ordenados y comparan su rendimiento con otros algoritmos de ordenamiento, ofreciendo un análisis detallado de su complejidad.
- **Unai López de la Fuente** en "Algoritmos y estructuras de datos en Python" implementa el algoritmo en Python, proporcionando ejemplos prácticos y demostrando su uso en situaciones reales. Además, el autor analiza cómo el Insertion Sort puede ser optimizado en ciertos casos y dónde es más efectivo comparado con otros algoritmos.
- **Gustavo López et al.** en "Análisis y diseño de algoritmos" muestran cómo implementar el algoritmo en lenguajes como C y Pascal. El enfoque es práctico, con ejemplos de código que detallan cómo insertar cada elemento en su posición correcta dentro de la lista ya ordenada, destacando su utilidad en contextos educativos y aplicaciones con listas pequeñas.

Ejemplo de Pseudocódigo del Algoritmo de Inserción

```
Algoritmo InsertionSort(lista)
   Para i desde 1 hasta longitud(lista) - 1 hacer
       valorActual ← lista[i]
       j ← i - 1
       // Desplazar los elementos de la lista que son mayores que valorActual
       // a una posición adelante de su posición actual
       Mientras j >= 0 y lista[j] > valorActual hacer
            lista[j + 1] ← lista[j]
            j ← j - 1
       FinMientras
       // Insertar el valorActual en su posición correcta
       lista[j + 1] ← valorActual
    FinPara
FinAlgoritmo
```

Ordenamiento por Inserción | Insertion sort
Python - Nivel 31 - Reto 3 - Ordenamiento de burbuja

Descripción del Pseudocódigo

1. **Entrada**: El algoritmo recibe una lista de n elementos que se desea ordenar.
2. **Iteración Principal**: Se recorre la lista desde el segundo elemento hasta el último. Se considera que el primer elemento ya está ordenado.
3. **Desplazamiento de Elementos**: Para cada elemento, se comparan los elementos anteriores en la lista ya ordenada. Si son mayores que el elemento actual, se desplazan una posición hacia la derecha.
4. **Inserción**: Una vez que se encuentran elementos menores o iguales, se inserta el elemento actual en su posición correcta.
5. **Salida**: Al finalizar todas las iteraciones, la lista está ordenada en orden ascendente.

Ordenamiento por Selección (Selection Sort)

El **algoritmo de ordenamiento por selección** (conocido en inglés como **Selection Sort**) es un método de ordenamiento simple que divide la lista en dos partes: una parte ordenada y otra desordenada. El algoritmo selecciona repetidamente el elemento mínimo (o máximo, dependiendo del orden) de la parte desordenada y lo coloca al final de la parte ordenada. Este proceso se repite hasta que toda la lista está ordenada. Aunque su complejidad temporal es $O(n^2)$, lo que lo hace ineficiente para listas grandes, es fácil de entender y de implementar, siendo útil en ciertos contextos donde la simplicidad es clave.

- **Luis Joyanes Aguilar** en "Fundamentos de Programación" describe el algoritmo de selección como un método simple pero con una eficiencia limitada. Presenta el algoritmo con un pseudocódigo claro, explicando cómo se selecciona el elemento más pequeño de la parte desordenada y se intercambia con el primer elemento de esta parte, repitiendo el proceso hasta ordenar la lista completa.
- **Thomas H. Cormen et al.** en "Introduction to Algorithms" explican el algoritmo desde un punto de vista teórico, enfocándose en su simplicidad y facilidad de implementación. Discuten cómo el Selection Sort realiza un número constante de

intercambios, pero es menos eficiente en términos de comparaciones, lo que lo hace menos competitivo frente a otros algoritmos de ordenamiento.

- **Unai López de la Fuente** en "Algoritmos y estructuras de datos en Python" implementa el Selection Sort en Python, mostrando su uso práctico y proporcionando ejemplos claros. El autor discute cómo el algoritmo puede ser útil en listas pequeñas o cuando los recursos computacionales son limitados.
- **Gustavo López et al.** en "Análisis y diseño de algoritmos" presentan implementaciones del algoritmo en C y Pascal, mostrando cómo se puede codificar de manera práctica. Se enfatiza su uso como una herramienta educativa para entender los fundamentos de los algoritmos de ordenamiento.

Ejemplo de Pseudocódigo del Algoritmo de Selección

```
Algoritmo SelectionSort(lista)
    Para i desde 0 hasta longitud(lista) - 2 hacer
      // Suponer que el primer elemento no ordenado es el mínimo
         minIndice ← i
      // Buscar en la sublista el elemento más pequeño
         Para j desde i + 1 hasta longitud(lista) - 1 hacer
             Si lista[j] < lista[minIndice] entonces
                 minIndice ← j
             FinSi
         FinPara
        // Intercambiar el elemento más pequeño encontrado con el primero no ordenado
         Si minIndice ≠ i entonces
             temp ← lista[i]
             lista[i] ← lista[minIndice]
             lista[minIndice] ← temp
         FinSi
    FinPara
FinAlgoritmo
```

Descripción del Pseudocódigo

1. **Entrada**: El algoritmo recibe una lista de n elementos que se desea ordenar.
2. **Iteración Principal**: Se recorre la lista desde el primer hasta el penúltimo elemento, ya que el último estará automáticamente en su lugar correcto después de la última iteración.
3. **Selección del Mínimo**: Para cada posición en la lista, se busca el elemento más pequeño en la sublista desordenada (a partir de la posición actual).
4. **Intercambio**: Si se encuentra un elemento más pequeño que el elemento actual, se intercambian sus posiciones.
5. **Salida**: Al final de todas las iteraciones, la lista estará ordenada en orden ascendente.

Python - Nivel 31 - Reto 2 - Algoritmo de ordenamiento por selección
Ordenamiento por Selección | Selection Sort

Algoritmo de Búsqueda Binaria (Binary Search)

El **algoritmo de búsqueda binaria** es un método eficiente para encontrar un elemento en una lista ordenada. A diferencia de la búsqueda lineal, que recorre secuencialmente todos los elementos, la búsqueda binaria divide repetidamente la lista en mitades y descarta la mitad en la que no puede estar el elemento buscado. Este proceso se repite hasta que se encuentra el elemento o la sublista se reduce a cero. La búsqueda binaria es muy eficiente con una complejidad temporal de O(log n), lo que la hace ideal para listas grandes que ya están ordenadas.

- **Luis Joyanes Aguilar** en "Fundamentos de Programación" describe la búsqueda binaria como un método que mejora significativamente la eficiencia en listas ordenadas. Presenta el algoritmo utilizando pseudocódigo y ejemplos prácticos, enfatizando la importancia de que la lista esté previamente ordenada para que el algoritmo funcione correctamente.
- **Thomas H. Cormen et al.** en "Introduction to Algorithms" exploran la búsqueda binaria en profundidad, discutiendo su complejidad y el análisis matemático que respalda su eficiencia. El libro también incluye variantes del algoritmo y lo compara con otros métodos de búsqueda, subrayando su utilidad en estructuras de datos ordenadas.
- **Unai López de la Fuente** en "Algoritmos y estructuras de datos en Python" proporciona una implementación en Python de la búsqueda binaria, destacando su aplicación práctica y mostrando cómo puede integrarse en programas reales. También se discuten las limitaciones del algoritmo, como la necesidad de listas ordenadas.
- **Gustavo López et al.** en "Análisis y diseño de algoritmos" presentan implementaciones del algoritmo en C y Pascal, explicando cómo codificarlo en estos lenguajes y destacando su utilidad en situaciones donde la eficiencia es crítica, especialmente en comparación con la búsqueda lineal.

Ejemplo de Pseudocódigo del Algoritmo de Búsqueda Binaria

```
Algoritmo BusquedaBinaria(lista, valorBuscado)
    inicio ← 0
    fin ← longitud(lista) - 1
    Mientras inicio ≤ fin hacer
        medio ← (inicio + fin) / 2  // Calcular el punto medio de la lista
        Si lista[medio] = valorBuscado entonces
            Retornar medio  // Se encontró el valor, retornar su índice
        FinSi
        Si lista[medio] < valorBuscado entonces
            inicio ← medio + 1  // Buscar en la mitad derecha
        Sino
            fin ← medio - 1  // Buscar en la mitad izquierda
        FinSi
    FinMientras
    Retornar -1  // El valor no se encontró en la lista
FinAlgoritmo
```

Descripción del Pseudocódigo

1. **Entrada**: El algoritmo recibe una lista ordenada de n elementos y el valor que se desea buscar.
2. **Inicialización**: Se definen dos variables, `inicio` y `fin`, que representan los límites actuales de la sublista donde se realiza la búsqueda.
3. **Bucle Principal**: Mientras `inicio` sea menor o igual a `fin`, el algoritmo continúa dividiendo la lista.
4. **Cálculo del Punto Medio**: Se calcula el punto medio de la sublista actual.
5. **Comparación**:
    - Si el elemento en el punto medio es igual al valor buscado, se retorna el índice.
    - Si el elemento en el punto medio es menor que el valor buscado, se ajusta `inicio` para descartar la mitad izquierda.
    - Si el elemento en el punto medio es mayor, se ajusta `fin` para descartar la mitad derecha.
6. **Terminación**: Si `inicio` supera a `fin`, significa que el valor no está en la lista y se retorna -1.
7. **Salida**: Al final, el algoritmo retorna el índice del valor encontrado o -1 si no se encuentra.

Python - Nivel 23 - Reto 6 - Búsqueda binaria